

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО
ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И
ОПТИКИ»**

Факультет безопасности информационных технологий
Кафедра проектирования и безопасности компьютерных систем

Дисциплина:
«Операционные системы»

**ОТЧЕТ О ВЫПОЛНЕНИИ
ЛАБОРАТОРНОЙ РАБОТЫ №6**

Выполнил:
N3247 Гаврилова В.В.

Проверил:
Ханов А. Р.

Санкт-Петербург
2021г.

Задание:

Часть 1

Протестировать функцию malloc/free и построить график зависимости времени выделения от размера запрашиваемой памяти.

Либо винда, либо линукс

Часть 2

Сравнить с другими малоками

Выполнение:

Часть 1

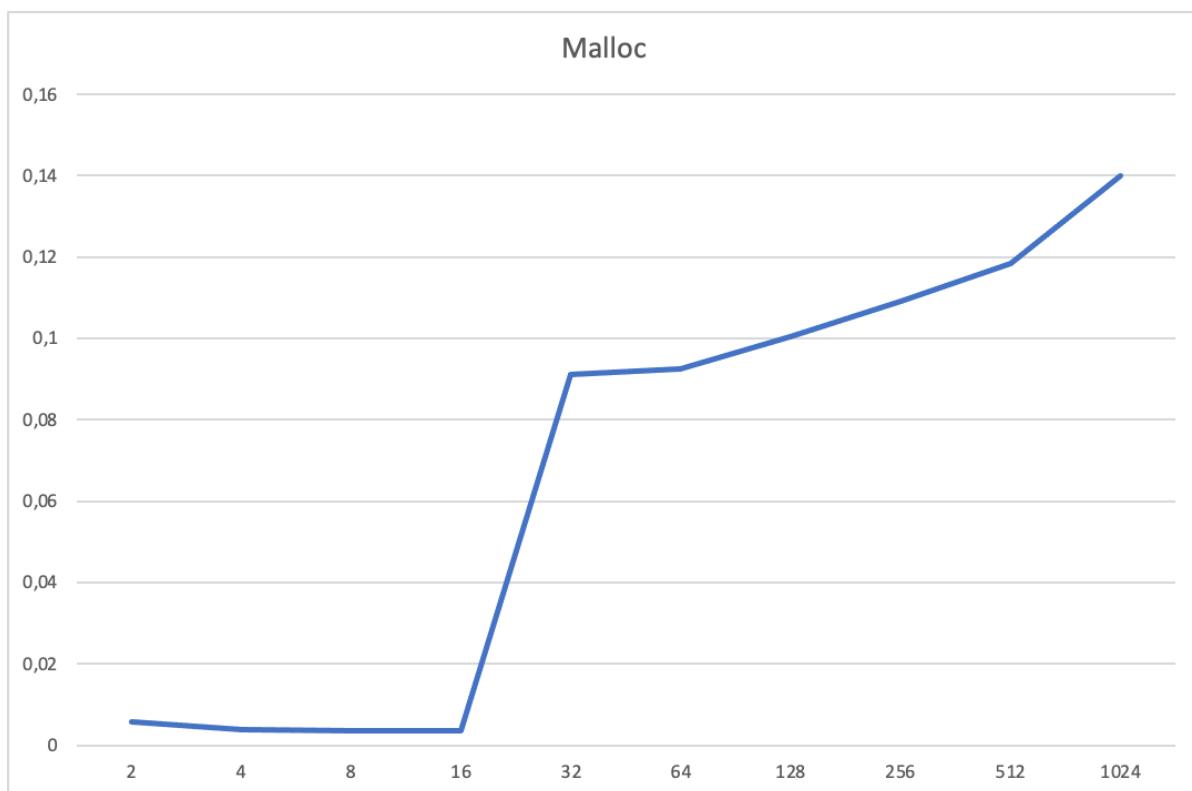
Создадим программу для тестирования malloc и calloc

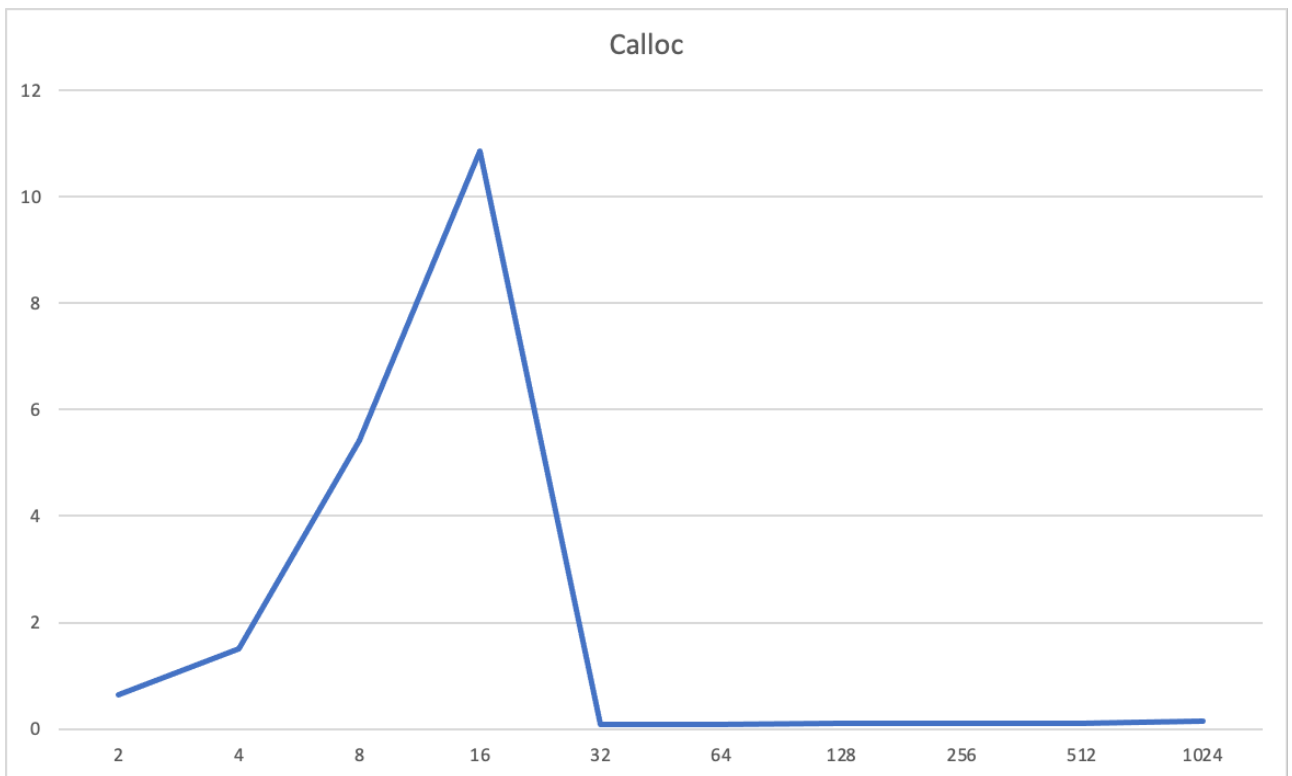
```
7 int main()
8 {
9     const int my_counter = 10;
10    const int my_test = 15000;
11    double timer1, timer2;
12    int i, a, k;
13    clock_t begin, end;
14
15    for(i = 1, a=2; i <= my_counter; i++, a*=2){
16        begin = clock();
17        for( k = 1 ; k < my_test; k++){
18            char *b = (char*) malloc(a*1024*1024);
19            b[0]=1;
20            free(b);
21        }
22        end = clock();
23        timer1 = (double)(end - begin)/CLOCKS_PER_SEC;
24        begin = clock();
25        for(k = 0; k < my_test; k++){
26            char *b = (char*) calloc(a, 1024*1024);
27            b[0]=1;
28            free(b);
29        }
30        end = clock();
31        timer2 = (double)(end - begin)/CLOCKS_PER_SEC;
32        printf("%dmb malloc: %f calloc: %f\n", a, timer1, timer2);
33    }
34 }
```

Получим

```
[root@parrot]-[/home/cyberknopa]
# ./a.out
2mb malloc: 0.005734 calloc: 0.646646
4mb malloc: 0.003758 calloc: 1.510209
8mb malloc: 0.003674 calloc: 5.430327
16mb malloc: 0.003504 calloc: 10.862459
32mb malloc: 0.091116 calloc: 0.086072
64mb malloc: 0.092615 calloc: 0.089273
128mb malloc: 0.100336 calloc: 0.104860
256mb malloc: 0.109121 calloc: 0.105622
512mb malloc: 0.118380 calloc: 0.114614
1024mb malloc: 0.140033 calloc: 0.138092
```

График:





Часть 2. Сравнение с другими маллоками

Для начала, давайте повторим подсчеты времени работы маллока стандартного из `stdlib.h` с помощью использования динамической библиотеки (обёртка маллока):

```
#define _GNU_SOURCE
#include <stdio.h>
#include <dlfcn.h>
#include <time.h>

static void* (*real_malloc)(size_t);
static void (*real_free)(void*);

unsigned long long int tim(){
    struct timespec t = {0, 0};
    clock_gettime(CLOCK_MONOTONIC, &t);
    return t.tv_sec*1000000000LL + t.tv_nsec;
}

static void trace_malloc(void){
    real_malloc = dlsym(RTLD_NEXT, "malloc");
    if (real_malloc == NULL)
        fprintf(stderr, "Error in dlsym: %s\n", dlerror());
    real_free = dlsym(RTLD_NEXT, "free");
    if (real_free == NULL)
        fprintf(stderr, "Error in dlsym: %s\n", dlerror());
}

void* malloc(size_t size){
    if (real_malloc == NULL)
        trace_malloc();
    void* ptr = NULL;
    unsigned long long int t1 = tim();
    ptr = real_malloc(size);
    unsigned long long int t2 = tim();
    fprintf(stderr, "malloc'd size=%d (%d MiB) || ptr=%p || time=%lld (nanosec)\n", size, size/(1024*1024), ptr, t2-t1);
    return ptr;
}

void free(void* p){
    unsigned long long int t1 = tim();
    real_free(p);
    unsigned long long int t2 = tim();
    fprintf(stderr, "free'd ptr=%p || time=%lld (nanosec)\n", p, t2-t1);
}
```

`gcc wrap.c -o wrap.so -shared -fPIC -ldl`

Малюсенький тест:

```
#include <stdio.h>
#include <stdlib.h>

int main(){
    char* ptr = NULL;
    for (int i = 0, a = 2; i < 10; i++, a*=2){
        ptr = (char*) malloc(a*1024*1024);
        ptr[0] = 1;
        free(ptr);
    }
}
```

gcc test.c -o t1

Результаты

```
root@kali:/mnt/hgfs/vm_share/labs/OS/lab6# LD_PRELOAD=`pwd`/wrap.so ./t1 2>&1 > /dev/null
malloc'd size=2097152 (2 MiB) || ptr=0x7f53d53a0010 || time=35358 (nanosec)
free'd ptr=0x7f53d53a0010 || time=7187 (nanosec)
malloc'd size=4194304 (4 MiB) || ptr=0x7f53d51a0010 || time=3878 (nanosec)
free'd ptr=0x7f53d51a0010 || time=2417 (nanosec)
malloc'd size=8388608 (8 MiB) || ptr=0x7f53d4da0010 || time=1835 (nanosec)
free'd ptr=0x7f53d4da0010 || time=1974 (nanosec)
malloc'd size=16777216 (16 MiB) || ptr=0x7f53d45a0010 || time=1616 (nanosec)
free'd ptr=0x7f53d45a0010 || time=1899 (nanosec)
malloc'd size=33554432 (32 MiB) || ptr=0x7f53d35a0010 || time=3822 (nanosec)
free'd ptr=0x7f53d35a0010 || time=2922 (nanosec)
malloc'd size=67108864 (64 MiB) || ptr=0x7f53d15a0010 || time=2043 (nanosec)
free'd ptr=0x7f53d15a0010 || time=2274 (nanosec)
malloc'd size=134217728 (128 MiB) || ptr=0x7f53cd5a0010 || time=1644 (nanosec)
free'd ptr=0x7f53cd5a0010 || time=2202 (nanosec)
malloc'd size=268435456 (256 MiB) || ptr=0x7f53c55a0010 || time=3205 (nanosec)
free'd ptr=0x7f53c55a0010 || time=3602 (nanosec)
malloc'd size=536870912 (512 MiB) || ptr=0x7f53b55a0010 || time=3839 (nanosec)
free'd ptr=0x7f53b55a0010 || time=5892 (nanosec)
malloc'd size=1073741824 (1024 MiB) || ptr=0x7f53955a0010 || time=3422 (nanosec)
free'd ptr=0x7f53955a0010 || time=5915 (nanosec)
```

Такой способ очень удобен, можно «на лету» перехватывать вызываемые функции и модифицировать их вызов (путём принудительной линковки). Однако, для дальнейшего тестирования это не подойдет.

Дело вот в чем: далее будут тестироваться следующие аллокаторы с открытым исходным кодом: jemalloc, tcmalloc, hoard. Эти аллокаторы представляют собой менеджеры кучи, оптимизированные под многопоточность, то есть, как заверяют их разработчики, должны эффективнее с кэшами (объемы памяти) и быстрее по времени. Какие ограничения это накладывает на тесты? Очевидно, что теперь нам нужно запускать много потоков, и мерить скорость выполнения маллока и эффективность работы с памятью. Здесь трюки с LD_PRELOAD не пройдут.

Итак, к делу.

Дизайн эксперимента: Kali Linux (виртуальная машина на VmWare Workstation 15), 8 Gib RAM, 3 cores (out of 6 cores of Intel Core I5 9600k). Будем выделять последовательно чанки размерами $a*65$ Kib, где a – степень двойки от 1 до 9. Затем будем заполнять весь чанк

каким-нибудь байтом при помощи `memset` (время исполнения `malloc` рассчитывается без `memset`). В это время считается максимальный Resident Set Size процесса в RAM (какое кол-во страниц процесса на данный момент реально существует в оперативной памяти, к которым можно будет обратиться без `page fault`, а то есть – без дополнительных затрат времени на выделение страницы). Бенчмарк будет запускать от 1 до 20 потоков. Код бенчмарка на языке C:

```
GNU nano 5.2
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
// #include <jemalloc/jemalloc.h>

#define ERROR_CREATE_THREAD -11
#define ERROR_JOIN_THREAD -12
#define SUCCESS 0

unsigned long long int tim(){
    struct timespec t = {0, 0};
    clock_gettime(CLOCK_MONOTONIC, &t);
    return t.tv_sec*1000000000LL + t.tv_nsec;
}

int parseLine(char *line) {
    char str[128];
    int i = strlen(line);
    const char *p = line;
    while (*p < '0' || *p > '9') p++;
    line[i - 3] = '\0';
    strncpy(str, p, 128);
    return atoi(str);
}

int getProcessMemory() {
    FILE *file = fopen("/proc/self/status", "r");
    char line[128];
    int rssInKb;

    while (fgets(line, 128, file) != NULL) {
        if (strncmp(line, "VmRSS:", 5) == 0) {
            rssInKb = parseLine(line);
            break;
        }
    }
    fclose(file);
    return rssInKb;
}

typedef struct arguments{
    int maxRSS;
    int totalAllocd;
    unsigned long long int ns_per_malloc_vol[9];
    unsigned long long int ns_per_free_vol[9];
} args_type;
```

```

void* allocator(void* args){
    args_type* arg = (args_type*) args;
    arg->maxRSS = 0;
    arg->totalAllocd = 0;
    char* ptr[9];
    for (int i = 0; i < 9; i++) ptr[i] = NULL;
    int size;
    unsigned long long int t1, t2;
    for (int i = 0, a = 2; i < 9; i++, a*=2){
        size = a*1024*13*5;
        t1 = tim();
        ptr[i] = (char*) malloc(size);
        t2 = tim();
        if (ptr[i] == NULL) printf("ERROR: can't alloc size %d", size);
        arg->ns_per_malloc_vol[i] = t2 - t1;
        memset(ptr[i], 'G', size);
        int cur_RSS = getProcessMemory();
        if (cur_RSS > arg->maxRSS) arg->maxRSS = cur_RSS;
        arg->totalAllocd += size/1024;
    }
    for (int i = 0; i < 9; i++){
        t1 = tim();
        free(ptr[i]);
        t2 = tim();
        arg->ns_per_free_vol[i] = t2 - t1;
    }
    return SUCCESS;
}

#define MAX_THREADS 20

int main(){
    pthread_t threads[MAX_THREADS+1];
    args_type args_per_thr[MAX_THREADS+1];
    int stat, stat_join;
    int rss_per_am_th[MAX_THREADS+1];

    for (int am_th = 1; am_th <= MAX_THREADS; am_th++){
        for (int i = 1; i <= am_th ; i++){
            stat = pthread_create(&threads[i], NULL, allocator, (void*) &args_per_thr[i]);
            if (stat != 0){
                printf("main error: can't create thread, status = %d\n", stat);
                exit(ERROR_CREATE_THREAD);
            }
        }
        for (int i = 1; i <= am_th ; i++){
            stat = pthread_join(threads[i], (void**)&stat_join);
            if (stat != 0){
                printf("main error: can't join thread, status = %d\n", stat);
                exit(ERROR_JOIN_THREAD);
            }
        }
        int totAlloc_bytes = 0;
        int MaxRss_Kb = 0;
        unsigned long long int max_ns_malloc[9];
        unsigned long long int max_ns_free[9];
        for (int i = 1; i <= am_th; i++){
            if (args_per_thr[i].maxRSS > MaxRss_Kb) MaxRss_Kb = args_per_thr[i].maxRSS;
            totAlloc_bytes += args_per_thr[i].totalAllocd;
            for (int j = 0; j < 9; j++){
                if (max_ns_malloc[j] < args_per_thr[i].ns_per_malloc_vol[j]) max_ns_malloc[j] = args_per_thr[i].ns_per_malloc_vol[j];
                if (max_ns_free[j] < args_per_thr[i].ns_per_malloc_vol[j]) max_ns_free[j] = args_per_thr[i].ns_per_malloc_vol[j];
            }
        }
        //long long int totTime = 0;
        printf("Amount of Threads: %d\n\n", am_th);
        printf("max malloc (ns), max free (ns)\n");
        for (int j = 0; j < 9; j++){
            printf("%llu,%llu\n", max_ns_malloc[j], max_ns_free[j]);
        }
        printf("\nMaxRSS (kb), total Allocd (kb)\n");
        printf("%d,%d\n\n\n", MaxRss_Kb, totAlloc_bytes);
    }
    return 0;
}

```

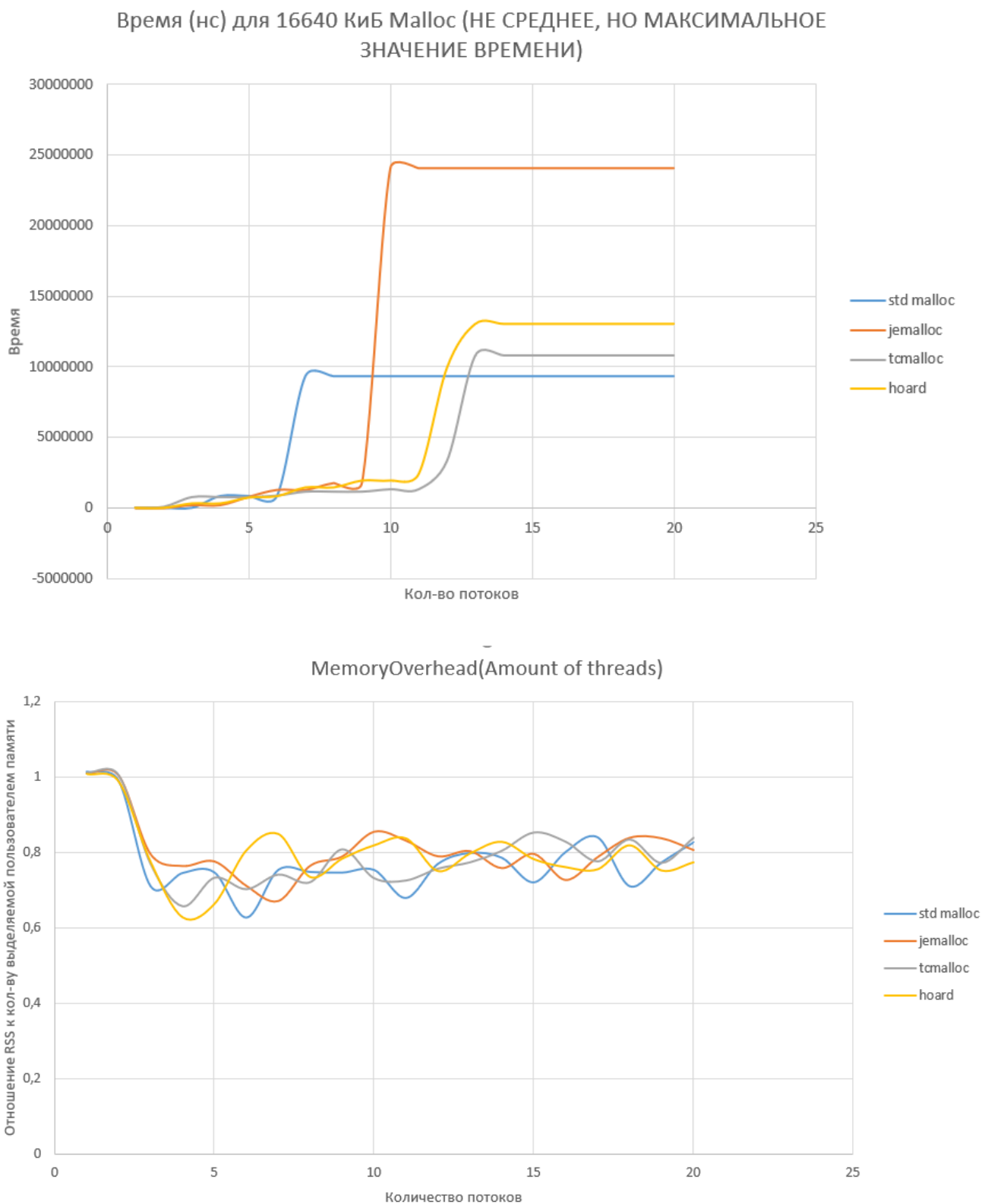
apt-get install google-perftools

apt-get install libjemalloc-dev

git clone <https://github.com/emeryberger/Hoard>; cd Hoard/src/ ; make

gcc -pthread -fno-builtin-malloc -fno-builtin-calloc -fno-builtin-realloc
-fno-builtin-free (-ljemalloc or -lhoard or -ltcmalloc) tester.c -o test

Результаты тестирования:



(книга Эксель находится вместе с отчетом в архиве – графики со страницы «результаты и графики»)

Краткий анализ:

Насколько можно видеть, максимальное время на маллок у стандартного начинает увеличиваться по сравнению с оптимизированными начиная с 6 потоков. Да, можно заметить, что начиная с 10 потоков максимальное время на маллок у «многопоточных» менеджеров кучи резко подскакивает вверх, превышая время стандартного, но на то оно и максимальное. Тем не менее, график свидетельствует о том, что вероятнее всего В СРЕДНЕМ tcmalloc, hoard и jemalloc будут работать значительно быстрее при числе потоков больше 6 (и вплоть до 20). По скорости лидер – tcmalloc.

Посмотрим на второй показатель – Memory overhead (иными словами «избыточность» использования памяти, как отношение максимального RSS к выделяемому приложением (пользователем) памяти). Здесь не так однозначно. Если смотреть с точки зрения эффективности работы конкретного процесса (выделяющего память) – в данном случае нашего – чем выше отношение, тем быстрее пользователь сможет обращаться к выделенной памяти (без page fault): с этой точки зрения лидером является jemalloc. Однако с точки зрения системы в целом – чем ниже этот показатель – тем больше у системы свободных страниц (в RAM) для других приложений (и прочего). Конкретно в моём тесте лучше всего себя показывают tcmalloc и hoard по этому показателю (более эффективно используют процессорные кэши, можно подумать).

P.S !! Данный тест очень отдаленно эмулирует работу с кучей реальных современных приложений, а посему результаты могут уводить в сторону от таковых в реальных кейсах. Однако, славно, что результат получился вменяемым и хоть сколько-то интерпретируемым (не без огрехов автора).